**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Always check the information your program receives, whether it comes from a user, another program, or a file. This helps make sure it’s safe and in the correct format. If you skip this step, someone could try to trick your program using harmful input, which could lead to crashes or security problems. |
| 1. Heed Compiler Warnings | When you write code, the compiler sometimes shows warnings to let you know something might go wrong. Don’t ignore these warnings, they're like little red flags. Fixing them early can help you avoid bugs and security issues later on. |
| 1. Architect and Design for Security Policies | When planning how your software will work, think about security from the start. It’s easier and safer to build strong protections into the design than to fix problems later. This includes things like who can access what, and how data is protected. |
| 1. Keep It Simple | The simpler your code is, the easier it is to understand, test, and keep safe. Complicated code is more likely to have mistakes that hackers can take advantage of. Write clean and clear code instead of trying to be fancy. |
| 1. Default Deny | Only allow access to things that are clearly safe and needed. If something isn’t specifically allowed, block it. This helps prevent unwanted actions or people from getting in by mistake. |
| 1. Adhere to the Principle of Least Privilege | Give each person or part of your program only the access it really needs—nothing more. This way, if something goes wrong, the damage is limited because no one has more power than necessary. |
| 1. Sanitize Data Sent to Other Systems | Before your program sends information to things like a database or another website, clean it up first. This helps stop hackers from sneaking in dangerous commands through the data. |
| 1. Practice Defense in Depth | Don’t rely on just one layer of protection. Use several safety measures, like passwords, input checks, and firewalls, so if one fails, others still help keep things safe. |
| 1. Use Effective Quality Assurance Techniques | Test your code often and carefully. Use things like code reviews and automated tools to catch mistakes early. This helps find problems before they turn into serious bugs or security issues. |
| 1. Adopt a Secure Coding Standard | Follow a set of safe coding rules when writing software. These rules help you avoid common mistakes that can lead to security problems. It keeps your code consistent and much safer. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Avoid unnamed namespaces in header files |

| **Noncompliant Code** |
| --- |
| This header file defines a variable inside an unnamed namespace. Each file that includes this header gets its own copy, leading to inconsistent or buggy behavior. |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE  namespace {  int v;  }  #endif // A\_HEADER\_FILE |

| **Compliant Code** |
| --- |
| The variable is declared as extern in the header and defined in a .cpp file, so only one copy exists in the entire program. |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE  extern int v;  #endif // A\_HEADER\_FILE |
| // a.cpp  #include "a.h"  int v; |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  **3. Architect and design for security policies**: plan the structure of the code so it avoids hidden or hard-to-find problems.  **9. Use effective quality assurance techniques**: check the code often with tools or reviews to catch mistakes early.  **10. Adopt a secure coding standard**: follow an agreed set of safe coding rules, like CERT, so all code is written consistently and securely. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Unnamed-namespace-header | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL59 |  |
| Clang | 3.9 | cert-dcl59-cpp | Checked by clang-tidy |
| CodeSonar | 9.1p0 | LANG.STRUCT.DECL.ANH | Anonymous Namespace in Header File |
| Helix QAC | 2025.2 | C++2518 |  |
| Klocwork | 2025.2 | MISRA.NAMESPACE.UNMD |  |
| LDRA tool suite | 9.7.1 | 286 S, 512 S | Fully implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-DCL59-a | There shall be no unnamed namespaces in header files |
| Polyspace Bug Finder | R2024b | CERT C++: DCL59-CPP | Checks for unnamed namespaces in header files (rule fully covered) |
| RuleChecker | 22.10 | unnamed-namespace-header | Fully checked |
| SonarQube C/C++ Plugin | 4.10 | UnnamedNamespaceInHeader |  |
| PVS-Studio | 7.38 | V1068 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not delete arrays through incorrect pointer types |

| **Noncompliant Code** |
| --- |
| An array of Derived objects is created, but it's deleted using a Base\* pointer. Even though Base has a virtual destructor, this still results in undefined behavior. |
| struct Base {  virtual ~Base() = default;  };  struct Derived final : Base {};  void f() {  Base\* b = new Derived[10];  delete[] b;  } |

| **Compliant Code** |
| --- |
| The array is created and deleted using the correct type, Derived\*, which avoids undefined behavior. |
| struct Base {  virtual ~Base() = default;  };  struct Derived final : Base {};  void f() {  Derived\* b = new Derived[10];  delete[] b;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  **3. Architect and design for security policies**: structure code so it avoids dangerous memory handling patterns.  **9. Use effective quality assurance techniques**: use tools and testing to catch type mismatches early.  **10. Adopt a secure coding standard:** follow safe C++ practices like CERT EXP51-CPP to prevent undefined behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | No | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -analyzer-checker=cplusplus | Checked with clang -cc1 or (preferably) scan-build |
| CodeSonar | 9.1p0 | ALLOC.TM | Type Mismatch |
| Helix QAC | 2025.2 | C++3166 |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-EXP51-a | Do not treat arrays polymorphically |
| Klocwork | 2025.2 | CERT.EXPR.DELETE\_ARR.BASE\_PTR |  |
| Parasoft Insure++ | — | Runtime detection |  |
| Polyspace Bug Finder | R2024b | CERT C++: EXP51-CPP | Checks for delete operator used to destroy downcast object of different type |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not pass null pointers to std::string functions |

| **Noncompliant Code** |
| --- |
| This code creates a std::string from std::getenv(), which can return a null pointer if the environment variable is missing. Passing that null to the string constructor causes undefined behavior. |
| #include <cstdlib>  #include <string>  void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| The pointer from getenv() is checked before creating the string. If it’s null, an empty string is used instead. |
| #include <cstdlib>  #include <string>  void f() {  const char\* tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  1. **Validate input**: check any pointer before using it to make sure it’s not null.  **3. Architect and design for security policies:** write code so it handles invalid or missing data safely.  **9. Use effective quality assurance techniques**: use automated checks and reviews to catch null pointer risks.  **10. Adopt a secure coding standard:** follow rules like CERT STR51-CPP to avoid null pointer dereferencing. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | No | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 9.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2025.2 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Klocwork | 2025.2 | NPD.CHECK.CALL.MIGHT / NPD.CHECK.CALL.MUST / NPD.CHECK.MIGHT / NPD.CHECK.MUST / NPD.CONST.CALL / NPD.CONST.DEREF / NPD.FUNC.CALL.MIGHT / NPD.FUNC.CALL.MUST / NPD.FUNC.MIGHT / NPD.FUNC.MUST / NPD.GEN.CALL.MIGHT / NPD.GEN.CALL.MUST / NPD.GEN.MIGHT / NPD.GEN.MUST / RNPD.CALL / RNPD.DEREF |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2024b | CERT C++: STR51-CPP | Checks for string operations on null pointer (partially covered) |
| Security Reviewer – Static Reviewer | 6.02 | shiftTooManyBits | Fully implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Exclude user input from format strings |

| **Noncompliant Code** |
| --- |
| This code builds a format string that includes user input and passes it directly to fprintf(). If the input contains format specifiers (like %x or %n), it can crash or exploit the program. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  void incorrect\_password(const char \*user) {  static const char msg\_format[] = "%s cannot be authenticated.\n";  size\_t len = strlen(user) + sizeof(msg\_format);  char \*msg = (char \*)malloc(len);  snprintf(msg, len, msg\_format, user);  fprintf(stderr, msg); // Caution: format string from user input  free(msg);  } |

| **Compliant Code** |
| --- |
| This version safely passes the fixed format string and user input separately, avoiding any chance of format string injection. |
| #include <stdio.h>  void incorrect\_password(const char \*user) {  static const char msg\_format[] = "%s cannot be authenticated.\n";  fprintf(stderr, msg\_format, user);  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  **1. Validate input:** check and sanitize any data from outside sources before using it in format strings.  **8. Defense in depth:** combine validation with safe coding patterns to reduce risk if one layer fails.  **10. Adopt a secure coding standard:** follow established rules like CERT IDS06-J to prevent format string vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Yes | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Detects trust and security errors (see Chapter 8) |
| Parasoft Jtest | 2024.2 | CERT.IDS06.VAFS | Ensure the correct number of arguments for varargs methods with format strings |
| Klocwork | 2025.2 | SV.EXEC / SV.EXEC.DIR / SV.EXEC.ENV / SV.EXEC.LOCAL / SV.EXEC.PATH | Implemented taint and execution path checks |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not use std::rand() for pseudorandom number generation |

| **Noncompliant Code** |
| --- |
| This code uses rand() to generate part of an ID string. The result is predictable and can be easily guessed by an attacker. |
| #include <stdio.h>  #include <stdlib.h>  enum { len = 12 };  void func(void) {  char id[len];  int r;  r = rand();  snprintf(id, len, "ID%-d", r);  } |

| **Compliant Code** |
| --- |
| This code replaces rand() with the more secure random() and properly seeds it using srandom() and timespec\_get(). |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  enum { len = 12 };  void func(void) {  char id[len];  int r;  struct timespec ts;  if (timespec\_get(&ts, TIME\_UTC) == 0) {  // Handle error  }  srandom(ts.tv\_nsec ^ ts.tv\_sec);  r = random();  snprintf(id, len, "ID%-d", r);  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  **3. Architect and design for security policies**: choose random number generators that are strong enough for your application’s needs.  **8. Defense in depth**: use secure algorithms and proper seeding to reduce predictability.  **10. Adopt a secure coding standard:** follow rules like CERT MSC50-CPP to avoid weak or predictable random numbers. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Yes | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | bad-function (AUTOSAR.26.5.1A) | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC50 |  |
| Clang | 4.0 (prerelease) | cert-msc50-cpp | Checked by clang-tidy |
| CodeSonar | 9.1p0 | BADFUNC.RANDOM.RAND | Use of rand |
| ECLAIR | 1.2 | CC2.MSC30 | Fully implemented |
| Helix QAC | 2025.2 | C++5028 |  |
| Klocwork | 2025.2 | CERT.MSC.STD\_RAND\_CALL |  |
| LDRA tool suite | 9.7.1 | 44 S | Enhanced enforcement |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-MSC50-a | Do not use the rand() function for generating pseudorandom numbers |
| Polyspace Bug Finder | R2024b | CERT C++: MSC50-CPP | Checks for use of vulnerable pseudo-random number generator (partially covered) |
| RuleChecker | 22.10 | bad-function (AUTOSAR.26.5.1A) | Fully checked |
| Security Reviewer – Static Reviewer | 6.02 | RTOS\_07 | Fully implemented |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Do not rely on side effects in unevaluated operands |

| **Noncompliant Code** |
| --- |
| This code uses a++ inside a sizeof, but the increment never occurs. The value of a remains unchanged, which may confuse the developer or cause logic bugs. |
| void f() {  int a = 14;  int b = sizeof(a++);  std::cout << a << ", " << b << std::endl;  } |

| **Compliant Code** |
| --- |
| The increment happens outside of the sizeof, so the side effect is guaranteed to execute. |
| void f() {  int a = 14;  int b = sizeof(a);  ++a;  std::cout << a << ", " << b << std::endl;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  **4. Keep it simple:** avoid code that relies on hidden or confusing behavior like side effects in unevaluated expressions.  **9. Use effective quality assurance techniques**: apply static analysis and code reviews to spot side effects in places where they don’t execute.  **10. Adopt a secure coding standard:** follow CERT EXP52-CPP to ensure consistent and predictable behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Yes | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | sizeof | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-EXP52 |  |
| Clang | 3.9 | -Wunevaluated-expression | Warns when side effects are inside unevaluated expressions |
| CodeSonar | 9.1p0 | LANG.STRUCT.SE.SIZEOF | Side Effects in sizeof |
| Helix QAC | 2025.2 | C++3240, C++3241 |  |
| Klocwork | 2025.2 | MISRA.SIZEOF.SIDE\_EFFECT |  |
| LDRA tool suite | 9.7.1 | 54 S, 133 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-EXP52-a/b/c/d/e | Multiple rules for avoiding side effects in unevaluated operands |
| Polyspace Bug Finder | R2024b | CERT C++: EXP52-CPP | Checks for side effects in logical operator operands |
| RuleChecker | 22.10 | sizeof | Partially checked |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| This code throws an exception from throwing\_func() without catching it anywhere. The program will call std::terminate() and exit unexpectedly. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func(); // no try-catch  }  int main() {  f(); // uncaught exception leads to std::terminate()  } |

| **Compliant Code** |
| --- |
| This version adds a try-catch block in main() to catch any exception, allowing the program to shut down cleanly. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  try {  f();  } catch (...) {  // Handle error or log before exiting  }  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  **3. Architect and design for security policies:** structure your code so exceptions are handled at all levels, avoiding uncontrolled termination.  **8. Defense in depth:** use catch-all handlers in critical entry points like main() or thread functions to reduce damage from unexpected errors.  **10. Adopt a secure coding standard:** follow CERT ERR51-CPP to ensure all exceptions are caught and handled properly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Yes | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all / early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| CodeSonar | 9.1p0 | LANG.STRUCT.UCTCH / PARSE.MBDH | Masked by handler / Masked by default handler |
| Helix QAC | 2025.2 | C++4035, C++4036, C++4037 |  |
| Klocwork | 2025.2 | MISRA.CATCH.ALL |  |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-ERR51-a / CERT\_CPP-ERR51-b | Always catch exceptions; ensure every thrown exception has a compatible handler |
| Polyspace Bug Finder | R2024b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (partially covered) |
| RuleChecker | 22.10 | main-function-catch-all / early-catch-all | Partially checked |
| Security Reviewer – Static Reviewer | 6.02 | C35 | Fully implemented |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Iterator | [STD-008-CPP] | Use valid iterator ranges |

| **Noncompliant Code** |
| --- |
| Here, the begin and end iterators are reversed. This causes std::for\_each to increment past the end, which is undefined. |
| #include <algorithm>  #include <iostream>  #include <vector>  void f(const std::vector<int> &c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| This version passes the iterators in the correct order, ensuring valid behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>  void f(const std::vector<int> &c) {  std::for\_each(c.begin(), c.end(), [](int i) { std::cout << i; });  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  **4. Keep it simple:** use clear, correct iterator ranges to avoid undefined behavior.  **8. Defense in depth:** validate iterator ranges and container ownership before use.  **10. Adopt a secure coding standard:** follow CERT CTR53-CPP to ensure only valid iterator ranges are used. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | No | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference |  |
| CodeSonar | 9.1p0 | LANG.MEM.BO | Buffer Overrun |
| Helix QAC | 2025.2 | C++3802 |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-CTR53-a / CERT\_CPP-CTR53-b | Do not use an iterator range that isn't really a range / Do not compare iterators from different containers |
| Polyspace Bug Finder | R2024b | CERT C++: CTR53-CPP | Checks for invalid iterator range (partially covered) |
| PVS-Studio | 7.38 | V539, V662, V789 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Stream | [STD-009-CPP] | Do not access a closed file |

| **Noncompliant Code** |
| --- |
| This code attempts to use stdout with printf() after it has been closed, leading to undefined behavior. |
| #include <stdio.h>  int close\_stdout(void) {  if (fclose(stdout) == EOF) {  return -1;  }  printf("stdout successfully closed.\n"); // Undefined behavior  return 0;  } |

| **Compliant Code** |
| --- |
| The code safely avoids using stdout after it has been closed by redirecting output to stderr. |
| #include <stdio.h>  int close\_stdout(void) {  if (fclose(stdout) == EOF) {  return -1;  }  fputs("stdout successfully closed.\n", stderr); // Safe  return 0;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  **4. Keep it simple:** avoid using file pointers after they are closed to prevent unexpected errors.  **8. Defense in depth:** check resource states before use and redirect output if necessary after closing a stream.  **10. Adopt a secure coding standard:** follow CERT FIO46-C to prevent undefined behavior from use-after-close. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | No | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | Supported | Detects use of closed streams |
| CodeSonar | 9.1p0 | IO.UAC | Use after close |
| Coverity | 2017.07 | USE\_AFTER\_FREE | Implemented |
| LDRA tool suite | 9.7.1 | 48 D | Partially implemented |
| Helix QAC | 2025.2 | DF2696, DF2697, DF2698 |  |
| Klocwork | 2025.2 | SV.INCORRECT\_RESOURCE\_HANDLING.URH |  |
| Parasoft C/C++test | 2024.2 | CERT\_C-FIO46-a | Do not use resources that have been freed |
| PC-lint Plus | 1.4 | 2471 | Fully supported |
| Polyspace Bug Finder | R2024b | CERT C: Rule FIO46-C | Checks for use of previously closed resource (partially covered) |
| SonarQube C/C++ Plugin | 3.11 | S3588 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Thread Safety | [STD-010-CPP] | Wrap functions that can spuriously wake up in a loop |

| **Noncompliant Code** |
| --- |
| If the condition is not rechecked after the thread wakes up, the program may behave unpredictably, causing race conditions, consuming invalid data, or crashing. |
| if (list.next == nullptr) {  condition.wait(lock); // unsafe if woken up spuriously  } |

| **Compliant Code** |
| --- |
| This code safely uses a while loop to wrap the call to condition.wait(lock). This ensures the thread rechecks the condition predicate after waking up, preventing it from continuing prematurely in case of a spurious or accidental notification. This avoids race conditions or consuming data that is not ready. |
| while (list.next == nullptr) {  condition.wait(lock); // safe: rechecks the predicate  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  **3. Architect and design for security policies:** design thread synchronization so it safely handles spurious or malicious wakeups.  **8. Defense in depth**: always use a loop with condition variable waits to validate the predicate after waking.  **10. Adopt a secure coding standard:** follow CERT CON54-CPP to ensure condition variable waits are used correctly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Yes | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | LANG.STRUCT.ICOL / CONCURRENCY.BADFUNC.CNDWAIT | Inappropriate Call Outside Loop / Use of Condition Variable Wait |
| Helix QAC | 2025.2 | C++5019 | Helix QAC |
| Klocwork | 2025.2 | CERT.CONC.WAKE\_IN\_LOOP | Klocwork |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-CON54-a | Wrap functions that can spuriously wake up in a loop |
| Polyspace Bug Finder | R2024b | CERT C++: CON54-CPP | Detects condition variable waits not wrapped in loops |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation will be built into each stage of the DevSecOps process shown in the diagram to make sure our coding standards are always followed without relying only on manual checks. In the pre-production phase, tools will automatically scan for security issues during design, building, and testing, such as checking code for known risks, scanning for vulnerabilities, and verifying that only trusted components are used. When moving to production, automation will help with setting up security configurations, running health checks, and carrying out penetration tests. Once live, systems will automatically monitor logs and alerts to detect threats, respond by blocking attacks or rolling back changes, and regularly check that security settings are still in place. This continuous automation ensures Green Pace can catch problems early, fix them quickly, and keep systems secure without slowing down development.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Low | Unlikely | No | No | P1 |
| STD-003-CPP | High | Likely | No | Yes | P18 |
| STD-004-CPP | Medium | Unlikely | Yes | No | P4 |
| STD-005-CPP | Medium | Unlikely | Yes | No | P4 |
| STD-006-CPP | Low | Unlikely | Yes | Yes | P3 |
| STD-007-CPP | Low | Probable | Yes | Yes | P6 |
| STD-008-CPP | High | Probable | No | No | P6 |
| STD-009-CPP | Medium | Unlikely | No | No | P2 |
| STD-010-CPP | Low | Unlikely | Yes | No | P2 |

### 

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | All sensitive or confidential data stored on company systems, including databases, hard drives, cloud storage, and backups, must be encrypted using AES-256 or an equivalent strong encryption standard. Full disk encryption and data loss prevention (DLP) tools must be implemented to protect against unauthorized access. This applies to all endpoints, servers, and storage locations, including mobile devices, to ensure that stolen or misplaced devices do not expose data (Djalovic, 2025). |
| Encryption in flight | All data transmitted between systems, applications, or users must use secure communication protocols such as TLS 1.2+ or HTTPS, with strong ciphers and valid certificates from a trusted Certificate Authority. Email encryption must be applied to protect message content and attachments, and secure file transfer methods must be enforced. DLP policies must be active to detect and prevent sensitive data leaks during transmission, reducing interception risks (Djalovic, 2025) |
| Encryption in use | Data actively being accessed or processed must be protected using access controls, identity verification, and digital rights management to prevent unauthorized viewing or manipulation. Sensitive data in memory should be handled securely to avoid plain-text storage in temporary files, logs, or crash dumps. Hardware-based protections, such as Trusted Execution Environments, should be enabled when available. This limits exposure during processing, when data is most vulnerable (Djalovic, 2025). |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All users, devices, and services accessing company systems must be authenticated using secure methods, including multi-factor authentication (MFA) for user logins. Passwords must be hashed using strong, irreversible algorithms (e.g., Argon2id) and never stored in plain text. Service-to-service authentication should use secure tokens or certificates. This ensures that only verified identities can access protected resources |
| Authorization | Access rights must follow the principle of least privilege, using role-based or attribute-based access control (RBAC/ABAC). User permissions must be reviewed regularly, and any changes must be logged and approved. This policy applies to database queries, file access, and system configuration changes, ensuring that authenticated users cannot access resources beyond their role’s requirements |
| Accounting | All user and system activities affecting sensitive data must be logged and monitored. Logs must record user logins (successful and failed), database changes, addition or removal of users, permission modifications, and file access. Logs must be time-synchronized, stored securely, protected from tampering, and retained for at least one year. This enables traceability, supports forensic investigations, and ensures compliance with legal and regulatory requirements |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

**Audit Controls and Management**

This rule follows Principles 1, 2, 4, 7, and 8. Principle 1 (Least Privilege) means only the right people can get to or change audit records, so they stay safe. Principle 2 (Separation of Duties) makes sure no one person does everything in the process, which helps prevent mistakes or dishonesty. Principle 4 (Auditability) matches this rule directly, because it’s all about keeping good records and proof of compliance. Principle 7 (Defense in Depth) means there are several layers of checks to make sure the records are protected. Principle 8 (Accountability) makes sure all actions can be linked back to a person, so records actually have meaning.

**Enforcement**

This rule follows Principles 3, 4, 8, and 9. Principle 3 (Security by Design) means we build in ways to enforce the rules from the start, not as an afterthought. Principle 4 (Auditability) makes it possible to create reports every month for review. Principle 8 (Accountability) makes sure we can see who is responsible if someone breaks the rules. Principle 9 (Continuous Improvement) reminds us to keep our enforcement methods updated as things change.

**Exceptions Process**

This rule follows Principles 1, 2, 5, 8, and 9. Principle 1 (Least Privilege) means that even if someone gets an exception, it’s only for what they truly need. Principle 2 (Separation of Duties) means more than one person approves the exception so there’s no bias. Principle 5 (Risk Management) is about checking the risks before agreeing to an exception. Principle 8 (Accountability) makes sure all exceptions are written down with the names of those responsible. Principle 9 (Continuous Improvement) means exceptions are checked later to make sure they’re fixed on time.

**Distribution**

This rule follows Principles 6, 8, and 9. Principle 6 (Security Awareness and Training) fits here because every staff member must review and agree to the policy each year. Principle 8 (Accountability) makes sure we know who has read and accepted the policy. Principle 9 (Continuous Improvement) reminds us to update training and awareness when things change.

**Policy Change Control**

This rule follows Principles 5, 6, 9, and 10. Principle 5 (Risk Management) means we change the policy when new risks or compliance needs come up. Principle 6 (Security Awareness and Training) means staff will be told when the policy changes. Principle 9 (Continuous Improvement) is about keeping the policy fresh and up to date. Principle 10 (Compliance with Laws and Regulations) means the policy always matches current laws and industry rules.

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| [Insert text.] | 07/18/2025 | Milestone 3 | Cristian Chalarca |  |
| [Insert text.] | 08/10/2025 | Project 1 | Cristian Chalarca |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |

References:

Dinic, M. (2022, May 2). Encryption: Data at Rest, Data in Motion and Data in Use. Jatheon. <https://jatheon.com/blog/data-at-rest-data-in-motion-data-in-use/>

https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices